Reasoning about non-functional properties using compiler intrinsic function annotations

Shashank Jadhav
Hamburg University of Technology
shashank.jadhav@tuhh.de

Mikko Roth
Hamburg University of Technology
mikko.roth@tuhh.de

Heiko Falk
Hamburg University of Technology
heiko.falk@tuhh.de

Christopher Brown
University of St Andrews
cmb21@st-andrews.ac.uk

Adam Barwell
University of St Andrews
adb23@st-andrews.ac.uk

ABSTRACT

Embedded systems often need to adhere to time and energy constraints. With the increasing popularity of embedded systems, the interest in evaluating and optimizing non-functional properties like execution time and energy of these systems is increasing.

In this paper, we describe a Resource-usage Estimate Expression Language (REEL), which allows the user to argue about these properties, within the source code, in a compiler understandable manner. Furthermore, we discuss the integration of REEL within a compiler framework. We, also show the propagation of REEL annotations within the compiler, and how they can be exploited to make decisions based on the non-functional properties within the source code. Finally, we explore REEL's potential to perform ILP-based optimizations.

Keywords

Compilation, Annotations, Non-functional Properties, Function Inlining

1. INTRODUCTION

Modern embedded systems, often, are subjected to constraints like time and energy consumption. Depending upon the physical and functional aspects of these systems, the severity of the correctness of these systems may vary. Therefore, the non-functional properties like time and energy consumption are needed to be taken into consideration while developing for these systems. For embedded systems that are categorized as real-time, it is paramount that they should meet their deadlines. As embedded systems are becoming more common, there is an effort to strive for reliable and energy-efficient systems. Generating an energy-efficient code that guarantees a response within the specified time constraint or deadline is a way to tackle this issue.

Common compilers are likely to perform optimizations, which can lead to reasonably optimized code, but without any guarantees on execution times. When it comes to embedded systems, this often is not sufficient. We have to take into consideration fundamental properties like, e.g., the Worst-Case Execution Time (WCET). If we can ensure the WCET of the code is within a specified deadline, we can guarantee the real-time nature of an embedded system. If such properties can be estimated by static analysis methods, that are difficult to perform before compiling for a particular platform with appropriate optimizations enabled, they should be integrated into the compilation process. The compiler can then make correct decisions depending on such properties, allowing more generic code designs.

In this paper, we introduce the Resource-usage Estimate Expression Language (REEL), which is a collection of compiler intrinsic functions. It enables transparency and easy access to information about non-functional properties provided by the compiler from within the users' source code. It allows the user, or another tool for that matter, to reason about non-functional properties using the compiler's capabilities.

```
1 /* The WCET of this for-loop is important. */
2 for ( i = 0; i < 100; ++i )
3   a +=2;
4 /* This application should further run in
       either energy-efficient or high-performance
        mode, depending on the for-loop's WCET. */
```

Consider the code snippet above, in which the WCET of the for-loop is crucial for the user to make decisions about the applications' execution mode. In such cases, having access to the non-functional properties of the code is essential. Therefore, it would be useful if the user can annotate the code to steer the compiler to generate a suitable binary and provide the essential data.

We will consider REEL and its ongoing implementation within an existing compiler framework from the embedded system, in this case, the WCET-aware C Compiler (WCC). We will also explore the optimization potential, that the user can exploit during compilation by using REEL annotations within the source code.

This paper is organized as follows: Section 2 provides an overview of the related work and background on the compiler framework used. Section 3 describes the semantics and implementation of REEL Annotations. Section 4 talks about the integration of REEL annotations within WCC. Section 5 follows two different REEL annotations within WCC. We close with a conclusion and a brief look at future work.

2. BACKGROUND

2.1 Contract Specification Language

A Contract Specification Framework *Drive* is introduced by Brown et al. [2], which includes an Embedded Do-

main Specific Language, the Contract Specification Language (CSL). CSL allows the user to reason about the energy, time, and security properties of their code in an architecture-independent way. Reasoning about the worst, best, and average cases are possible for both time and energy cases. In case of security, CSL allows capturing the security level concerning fault injection, and power- and time-related side-channel attacks. The user needs to annotate their source code with predefined CSL-functions, that have to be placed before a regular C-statement to reason about the statement's non-functional properties.

The *Drive* framework can call external tools, such as WCC, for obtaining the necessary estimations and information about non-functional properties of the source code. REEL is being developed in conjunction with the University of St. Andrews, to have a usable interface between the two frameworks.

2.2 WCET-Aware C Compiler Framework

The WCET-Aware C Compiler (WCC) [4] is a C compiler consisting of sophisticated WCET-oriented analyses and optimizations. It currently supports the ARM7TDMI from ARM and TriCore TC1796 and TC1797 processors from Infineon. WCC is currently being extended to support the ARM Cortex-M0 and Leon3 from Cobham Gaisler.

WCC has two intermediate representations (IRs): one high-level C-like IR (ICD-C) and another assembler-like low-level IR (ICD-LLIR). The ICD-C framework is a data structure that provides a machine-independent IR for C code. On the other hand, ICD-LLIR is a data structure that provides a retargetable machine-dependent low-level IR for compiler back-ends. Various optimizations can be carried out on each IR-level. WCC offers WCET-aware optimizations during the compilation process due to its tight integration with a static WCET analyzer tool called *aiT* [1].

Additionally, WCC can perform WCET- and Energy-aware multi-objective optimizations. It can, also, perform ILP-based optimizations, to find optimal solutions that are derived deterministically, as described in, e.g., [5].

3. REEL ANNOTATIONS

CSL, a high-level language, is independent of any tool or compiler, and therefore a source code annotated with CSL is architecture-independent. To reason about physical and architecture-specific properties, such as energy, WCET, etc., it needs values or estimations for specific architectures. On the contrary, REEL is a collection of compiler intrinsic C-functions used to annotate C source code and provides estimations for specific architectures. REEL annotations sit between CSL and compiler level, offering an interface for obtaining such properties automatically in the future, as envisioned in [2].

REEL annotations enable the user to argue about the non-functional properties of the code. It is done by adding calls to the compiler intrinsic functions before regular C statements, much in the same way as is done in CSL. The user will be able to make decisions within their code, based on the nature of the non-functional properties. This information is injected into C-variables defined in the REEL function's argument-list, wherever it is appropriate. The user can also declare deadlines or limits on the energy consumption of a particular piece of code, by asserting within the source code. These REEL assertions will be treated as constraints while performing ILP-based optimization.

These annotations are expressed as regular ANSI-C compliant function calls in the source code. The user can annotate the source code with these REEL functions, defined in the REEL header file, which describes the meaning of each REEL annotation and provides information on their argument types. The function bodies for REEL annotations that provide their implementations are defined locally within the compiler. The user can inform the compiler about the important parts of the code in terms of non-functional properties using REEL annotations. For example, the following REEL annotations can inform the compiler that the user requires the WCET value of the code snippet immediately after the annotation.

```
1 void __reel_worst_time( unsigned long long *var
      );
```

Here, the variable `var` always refers to the WCET of the immediate next non-REEL C statement. If a user has some information about the non-functional properties of the code snippet, then the user can use the following annotation to inform the compiler about their values and the confidence level with which the user knows this information.

```
1 void __reel_worst_time_manual(
2                 unsigned long long *var,
3                 unsigned long long dValue,
4                 const _Bool cLevel );
```

Depending upon the confidence level `cLevel`, this annotation assigns either the user-provided default WCET value or a WCET value derived by a static WCET analysis to the variable `var`. Similar annotations can be used by the user if the user needs information in regards to the energy consumption, average-case execution time (ACET), etc. of the code snippets. Therefore, these values obtained by performing analyses can be used to make decisions within the application. These values should not be used within the analyzed code section itself, as it makes the analysis result dependent on itself.

Furthermore, `__reel_assert` provides C-level contracts to assert non-functional properties during compile-time.

```
1 void __reel_assert( _Bool expr );
```

This annotation provides a boolean C expression `expr`, which should be asserted by the compiler. The expression to be asserted is treated as integer-linear inequations within the compiler, which are never translated into actual machine code, contrary to other REEL annotations. This can help the user to deal with objectives like time and energy consumption for embedded systems at the source-code level. The user can define deadlines for embedded systems, which can be treated as integer-linear constraints while performing ILP-based optimizations within a compiler like WCC.

4. INTEGRATING REEL ANNOTATIONS INTO WCC

In this section, we describe the approach to realize REEL annotations within the WCC compiler framework, cf. also Figure 1. As mentioned before, WCC treats REEL annotations as compiler-known functions. The function bodies for REEL annotations that provide their implementations are defined locally within WCC in a dedicated C file. This

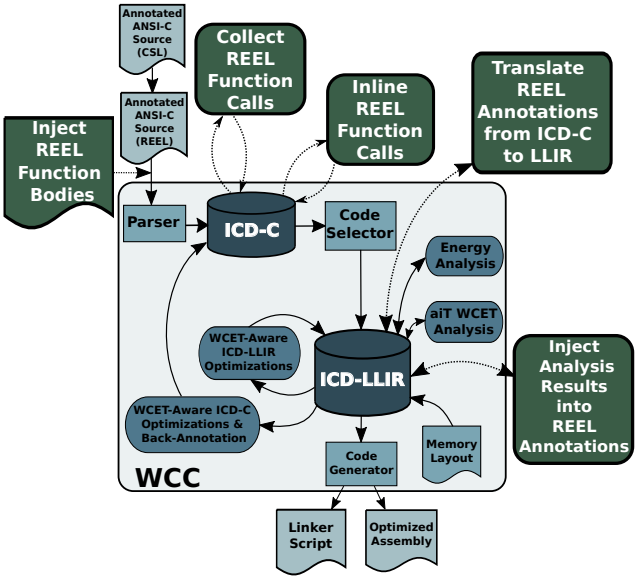

Figure 1: Integration of REEL Annotations within WCC

C file is transparently injected during WCC's compilation process, in the beginning, before the source files are parsed, as shown in Figure 1.

By using WCC's high-level intermediate representation ICD-C, REEL annotations within the source code are collected. To link these REEL annotation calls with their actual compiler-internal implementations, WCC inlines REEL function templates into which the analysis values are later injected. This way, all occurrences of REEL function calls are replaced by their respective function bodies, whose code is already injected previously within WCC using the C file.

Then ICD-C is translated into LLIR during WCC's code selection stage. For Leon and ARM architectures, WCC delegates code selection to external GCC instances, while for TriCore an internal implementation is used. As the external GCC is fully unaware of REEL annotations and WCC's internal mechanisms, it is very difficult to maintain a proper translation between the C-source level and the assembly-level. To tackle this issue, WCC exploits DWARF-2 debugging information [3]. DWARF-2 is a standardized extension of the binary object code that enables symbolic, source-level debugging. It provides an inherent mapping from assembly-level code to its original provenience at the source code-level. WCC propagates REEL annotations from ICD-C level to LLIR level by correctly exploiting the GCC-generated DWARF-2 debugging location information.

During code selection, it is necessary to properly identify the locations of code generated by the REEL function definitions. Therefore, WCC maintains a dictionary relating the ICD-C constructs to their LLIR counterparts. Every translation from an ICD-C basic block to an LLIR basic block is maintained with the help of a map. This way WCC can keep track of all the occurrences of these REEL annotations too at the assembly level. It is necessary to maintain this mapping so that the analyses for timing and energy can be performed at the assembly level.

WCC performs the required time and energy analyses at the LLIR level and annotates back this information to the LLIR at the granularity of individual basic blocks. Depending upon the scope of the REEL annotations within the LLIR, this information is aggregated such that it exactly covers the whole range of assembly-level code that the REEL annotation refers to at the source-level. For example, if a user requires information about the WCET value of a for-loop, WCC uses loop nesting to acquire a list of LLIR basic blocks associated with that for-loop and aggregates the WCET values of those basic blocks.

Once the aggregated data is collected at the LLIR level, WCC injects it into the assembly code that corresponds to the REEL construct and thus gets embedded into the binary. This data can also be exploited by the compiler optimizations, and thus allows the user to steer the compilation process.

5. EXAMPLE IMPLEMENTATION

5.1 REEL Worst Time

In this sub-section, we will consider a full implementation of one representative REEL annotation, i.e., `__reel_worst_time` defined in Section 3. WCC's internal implementation of this compiler-known REEL annotation looks as follows:

```
1 void __reel_worst_time( unsigned long long *var
      )
2 {
3   *var = 42;
4 }
```

This REEL annotation assigns WCET estimates to the variable `var`, given as a pointer parameter. A WCET value shall be assigned to this variable, which at first holds 42 as a dummy value. The WCET value assigned to this variable always refers to the WCET of the immediate next non-REEL C statement following the REEL annotation.

Consider the example C-source code from Section 1 annotated with a REEL annotation:

```
1 unsigned long long rWCET;
2 __reel_worst_time( &rWCET );
3 for ( i = 0; i < 100; ++i )
4   a +=2;
5 if ( rWCET < 10000 )
6   /* Execute energy-efficient mode. */
7 else
8   /* Execute high-performance mode. */
```

Here, a for-loop is annotated with a `__reel_worst_time` annotation. The value of variable `rWcet` is associated with the WCET of the following for-loop. Depending on the WCET value of the for-loop, the user is deciding either to perform an energy-efficient execution or to perform a high-performance execution. This code snippet is transformed via inlining into

```
1 unsigned long long rWCET;
2 *( &rWCET ) = 42;
3 for ( i = 0; i < 100; ++i )
4   a +=2;
5 if ( rWCET < 10000 )
6   /* Execute energy-efficient mode. */
7 else
8   /* Execute high-performance mode. */
```

If we are compiling this code for a ARM Cortex-M0 architecture, after code selection the above mentioned source code will be translated to the following assembly sequence:

```
.L1:
        mov     r6, #0          % b = 0
        mov     r3, #42         % rWCET = 42
        mov     r5, #0          % i = 0
        b       .L3
.L2:
        add     r6, r6, #2      % a += 2
        add     r5, r5, #1      % ++i
.L3:
        cmp     r5, #99         % i < 100
        bls     .L2
```

In the above assemble code snippet, the if-else statement is omitted for the sake of brevity. As it can be seen, the assembly-level basic block `.L1` features a `mov` instruction that corresponds to the original REEL annotation `__reel_worst_time( &rWCET );`. It assigns the dummy value of 42 to the register `r3` that holds the variable `rWCET`. This variable is supposed to represent the WCET of the entire for-loop in the above example. Thus, WCC performs a static WCET analysis of the generated assembly code by using *aiT*'s analyzer. If, for example, `aiT` determines that the WCET value of this for-loop is 420 clock cycles, WCC will finally update the `mov` instruction corresponding to the REEL annotation by

```
        mov     r3, #420        % rWCET = 420
```

Once this WCET value is replaced at the assembly code-level, this value can be accessed and used during run-time for various purposes like any other variable. In this example, the compiler could also discard the unused branch as soon as the value is known. Thus both cases can be taken into account during design time, without worrying about the properties of the final executable, as the compiler would make the right decision once the WCET has been determined. This makes the application more generic and easier to adjust for different target platforms or optimization levels.

5.2 REEL Assert

In this sub-section, we will consider a full implementation of `__reel_assert`, defined in Section 3. By using REEL assertion a user can provide a boolean C expression which is passed through WCC. WCC collects this boolean C expression and treats it as an integer-linear inequation.

A C-source code containing a REEL assert would look like as follows:

```
unsigned long long rWCET;
__reel_worst_time( &rWCET );
for ( i = 0; i < 100; ++i )
  a +=2;
/* The WCET value of the for-loop should be
    less than 10000. */
__reel_assert( rWCET < 10000 );
```

Unlike other REEL annotations, for `__reel_assert`, WCC collects the boolean expression and removes the assertion call from the source code. This boolean expression can be passed down to the ILP-based optimizations to be investigated in the near future. In the above-mentioned example, WCC collects the boolean expression `rWCET < 10000`. By using assert, the user is indicating that the WCET of the for-loop needs to be less than 10000. Once WCC performs an initial WCET analysis, the WCET value of the for-loop, associated with `__reel_worst_time`, is injected within the source.

Once `rWCET` is known, a linear-inequation is added to the ILP model as a constraint. The optimization then tries to find a solution that fulfills this assertion. If the assertion is not fulfilled by any possible solution, WCC will let the user know that the assertion is unrealistic, and a solution does not exist in such a scenario.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented REEL annotations, with which the user can argue about and obtain information about non-functional properties, like WCET, Energy, and security, at the source code level via annotations. It enables the user or other tools to obtain information about non-functional properties, and steer the compilation and optimization process. It also allows the user to make use of these non-functional properties, as the information is injected into user-defined variables to be used within the application.

We also discussed our implementation of REEL within a WCET-aware C compiler and showed an example implementation of two different REEL annotations. Furthermore, we showed that REEL assertions can be used by the user to provide constraints on the non-functional properties within the source code. These constraints will be taken into consideration by WCC while performing ILP-based optimizations, and WCC will try to find optimized solutions based on user-provided assertions.

As a next step, we will evaluate REEL annotations on a number of benchmarks and perform ILP-based optimizations to test our techniques. We plan to extend our work by taking into consideration other non-functional properties like energy. Our intention is also to extend the WCC and REEL framework to take security aspects of the code into consideration and see its behavior against WCET and Energy.

Acknowledgments

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779882.

7. REFERENCES

[1] G. AbsInt Angewandte Informatik. ait worst-case execution time analyzers, 2019.

[2] C. Brown, A. D. Barwell, Y. Marquer, C. Minh, and O. Zendra. Type-driven verification of non-functional properties. In *PPDP*, 2019. Accepted for publication.

[3] Dwarf debugging information format. `http://dwarfstd.org/`, 1993.

[4] H. Falk and P. Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.

[5] D. Oehlert, A. Luppold, and H. Falk. Bus-aware static instruction spm allocation for multicore hard real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.